JOURNAL OF SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE
J. Softw. Maint: Res. Pract. **11**, 143–150 (1999)

Opinion

Disposable Information Systems: The Future of Software Maintenance?†

JEFFREY M. VOAS*
Reliable Software Technologies Corporation, 21515 Ridgetop Circle, Sterling VA 20166–6509, U.S.A.

SUMMARY

News from the Information Technology (IT) community is mixed. Project backlogs, high employment and personnel shortages are claimed on one side; project cancellations, unproductive use of personnel time and continued poor quality software are claimed on the other. Which side is component-based software on? The drive to use components such as COTS components comes from claims of instant productivity gains, shorter time to market, lower cost, management mandates and a 'parts' philosophy. On the other side are difficulties in composing systems from unreliable components, the 'ility' factors, the traditional white-box software processes, short-changed testing, software component support limits and software malleability. If components such as COTS are used a lot, then finger-pointing grows more common as we move toward black-box component integration. Some of the defensive options include component certification, strengthened testing, and adding wrappers and middleware. All add to maintenance cost. If we use components such as COTS, our ability to maintain the resulting composite systems is reduced to the level of 'unmaintainable'. This leads us to think in terms of disposable information systems. Overall, I observe as we move toward component-based software, quality will be even more important for maintainability. Copyright © 1999 John Wiley & Sons, Ltd.

KEY WORDS: component-based software; COTS; disposable systems; software quality; software maintainability; software cost

1. THE STATE OF THE INDUSTRY

The news coming out of the Information Technology (IT) community today is mixed. The good news stems from predictions of continued industry growth. IT revenues for 1997 were USA$118.6 billion and are predicted to be USA$134.9 billion in 1998, USA$148.8 billion in 1999, USA$176.2 billion in 2000, USA$201.3 billion in 2001 and USA$230.9 billion in 2002 (IDC, 1998). This represents a near doubling of revenues in a five-year period. Also, corporate IT spending is up. In 1997, IT budgets were 6.9% of gross corporate revenue; in 1998, that figure is expected to be 7.1% (META, 1998a).

†This paper is a recasting and reorganization of Jeffrey M. Voas' keynote address 'Are COTS Products and Component Packaging Killing Software Malleability?' on 19 November 1998 at the 1998 International Conference on Software Maintenance (ICSM) in Bethesda MD, U.S.A.

*Correspondence to: Dr. Jeffrey M. Voas, Vice-President, Reliable Software Technologies Corporation, 21515 Ridgetop Circle, Suite 250, Sterling VA 20166–6509, U.S.A. Email: jmvoas@rstcorp.com

CCC 1040–550X/99/020143–08$17.50

Received 30 December 1998
Revised 15 January 1999

A serious worker shortage in the IT industry is the bad news. According to the ITAA's IT Workforce Convocation in January 1998, there are 340 000 unfilled IT jobs (Anderson *et al.*, 1997). Software Productivity Research estimates that the US shortage in software engineering personnel is between 100 000 and 300 000 people. They further go on to state that if you add in Year 2000 needs, that number may be as high as 700 000 (Jones, 1998b).

If you work in the IT industry, it may not be clear why this is problematic. After all, it does suggest 'high employability' without regard for qualifications. While that is ideal from a worker's perspective, the downside for the industry is that numerous new projects are never even started.

It is not clear, however, whether there is a real or artificial personnel shortage. An artificial personnel shortage occurs when, for example, people are on strike or people are wasting time while on the job. A real personnel shortage occurs when there is truly not enough manpower to complete waiting tasks. Jones (1998a) has analysed wasted personnel time in the IT industry. His analysis looked at how workers actually use their average 197 working days (this excludes weekends, vacations, holidays, sick days, training, department meetings—i.e., time on activities not directly tied to working on software) and found that testing and defect repairs accounted for 70 days, Y2K and related repairs accounted for 50 days, and time on projects that will ultimately be cancelled accounted for 30 days. This means then that 47 days, or 23.86% of the total worker time is applied to productive software activities. Note that if we could convert the 30 days spent on 'soon to be cancelled' projects, the 300 000+ worker shortage could be greatly reduced or even eliminated. I would argue that degree of time loss due to inefficiency is creating, to some degree, an artificial personnel shortage.

Further indications that our profession is in turmoil can be seen when we look at the quality of our products. Musa, Iannino and Okumoto (1987) exposed that defect densities have remained fairly constant during the past 20 years: 3–6 faults per KSLOC (thousand source-code lines of code). There are two interpretations that can made from this finding. We can argue that this is good news since systems are larger and more complex. Or we can argue a negative case since all of our advanced software engineering theory and tools have failed to decrease defect densities. Since the 3–6 faults per KSLOC figure was based on a fixed-size unit of code (as opposed to a per system basis), I would argue for the more negative position.

What we can summarize from the previous statistics are that:

1. approximately 75% of our time is spent on non-productive activities,
2. roughly two-thirds of the software workforce is engaged in rework and repairs,
3. 15% of the labour force is working on software systems that will not be deployed, and
4. there possibly would be no labour shortage if software quality could be brought under control.

We can speculate from these statistics that it is unlikely that any other major profession wastes so much effort.

From these statistics, the project cancellation figures are most troubling. Jones (1995) reports that 48% of all systems of size around 10 000 function points are never completed, and the numbers only get worse as the number of function points in a system increases. For example, Jones reports that 65% of all systems of size close to 100 000 function points are cancelled. The Chaos study by the Standish Group found similar statistics (Whiting, 1998). That study reported that 49% of projects whose costs were in excess of USA$10 million failed and for projects between USA$6 million and USA$10 million, the project failure rate was 41%.

While much of the aforementioned industry data seems pessimistic, it is not meant in that way. I am simply restating industry metrics. Further, we did not build skyscrapers after 40 years; it took thousands of years of engineering knowledge to reliably produce such physical systems.

Given the poor quality of software, what is surprising is the rapid move toward acquiring commercial-off-the-shelf (COTS) software from third-party vendors. In 1997, 25.5% of a typical corporation's IT portfolio was COTS software. In 1998, that figure is expected to jump to 28.5%. In 2002, this figure is expected to be 40% or greater (META, 1998b). Do we honestly believe that software developed elsewhere is better than if we developed it ourselves? Whatever happened to the adage 'If you want something done right you have to do it yourself'?

2. THE COTS QUAGMIRE

2.1. What is COTS software?

The are many definitions for what COTS software is. I use a very simple definition: COTS software is software functionality that is obtained from a third party and that is used on an 'as is' basis. Examples here include operating system utilities, class libraries, databases, word processing applications and browser plug-ins.

'COTS' for commercial-off-the-shelf, has become a buzzword that describes all purchased software, but there are other terms that have a similar interpretation:

- off-the-shelf (OTS);
- government-off-the-shelf (GOTS);
- non-developmental item (NDI)—this term is used by the FAA (Federal Aviation Administration) to differentiate software developed according to Standard DO-178B from that acquired otherwise (FAA, 1992);
- commercially acquired software (CAS)—this term is used by the FDA (Food and Drug Administration).

The commonality in all of these terms is that the software that they describe was acquired from elsewhere and usually will be delivered in executable format (as opposed to source-code format).

Thus, our profession is moving toward black-box component integration after decades of 'one-of-a-kind', white-box software development. That alone is not a concern until you couple it with the aforementioned quality deficiencies. Those problems become magnified for systems that are heavily comprised of COTS software because most of our quality improvement techniques will not be applicable by the user to the acquired software. These techniques are of course available to the software publisher, but whether the publisher opts to use them can rarely be independently verified by the user. That is alarming and could be highly problematic for organizations that opt to *buy-a-right-to-use* versus *build* software systems.

The notion of building software from bits and pieces of software is not new. For example, using a language-provided construct like '+' for addition fits the definition for COTS software. A function like cosine does as well. Languages like FORTRAN have provided that for years. So is the concern over COTS software well founded?

The fundamental difference between cosine utilities and component packaging is the level of granularity. When we programmed using language-defined functions, we were programming at a lower level and were likely to make more mistakes. Component-based development moves us up a level to the point where we are writing glue software instead of the parts being glued. The number of errors we will introduce into the system is less because we are writing less code. The problem is that we are gluing components that potentially have numerous defects, unknown side-effects and other behavioural anomalies. Hence, the finished product may well have as many defects as if we had developed the entire product ourselves. The issue, however, is that we will not have the ability to assess and debug the parts when components are used. We could if we had created the system from scratch.

All of this leads me to a position statement concerning quality, acquired software and maintainability:

> ***As we move toward component-based software engineering, quality will be even more important for maintainability.***

My belief is that our current levels of quality are too poor to make component-based systems viable, and our industry's long-standing lack of quality control will make it increasingly difficult to maintain component-based systems.

In fact, I believe that it will move us toward the same maintenance paradigm that we use today with personal computers: buy a new computer every two years instead of upgrading components (e.g., the processor). Why do we do this? The answer is simple: personal computers are no longer upgradable and they are cheap enough such that we do not view spending $2 000 every two years as unreasonable. The lack of maintainability afforded to the system integrator or user of COTS-based software systems will likely move us toward thinking more about *disposable* information systems as the solution to maintenance. Clearly, disposing of a system is a radical approach to maintenance, but if systems are truly unmaintainable, what other options are there?

2.2. Why COTS?

Let us begin by first looking at how we got to this situation. The shift away from custom software to acquired software has occurred as a result of five basic beliefs within the industry:

- *Instant productivity gains.* Because the industry average production rate for specified, tested and debugged code is still about 10 lines per person-day of effort (Baker, McWilliams and Kripalani, 1997), creating a one-million-line system from scratch could require years of calendar time. Acquiring large chunks of needed functionality immediately is tempting.
- *Time-to-market.* As little as a one month slippage in schedule can cost an organization enormous percentages in market share. Getting to market first is almost always the goal, regardless of initial quality.
- *Cost.* Estimates of costs to produce a line of debugged code range from $50 to $2 000. These figures have been backed out from total project costs and final number of lines of code in a delivered project. This great discrepancy in costs cover the spectrum from game software to safety-critical control code.

- *Mandates.* Organizations such as the DoD (US Federal Department of Defense) require the use of acquired components whenever possible (Perry, 1994). Many fear acquired software, but cannot argue (based on fact as opposed to intuition) why it should be avoided.
- *Philosophy.* There is a belief that we should build software systems in a manner similar to how hardware systems are built. Electrical engineers build systems around available components. They select as much as they can from catalogues. They then design the remainder of the system around what they have acquired and what they were unable to acquire and must build from scratch. Why not do the same for software systems?

Note that none of these advantages consider the impact of acquired software on the maintenance phase. Instead, quicker time-to-market and cheaper development costs are the criteria of interest.

Implicit in the ‘Philosophy’ justification is a hint that maintenance will someday be as simple as swapping software components in and out, like changing a flat tyre on an automobile (Gilb, 1977). Recognize, however, that changing a flat tyre begins by first making an assessment whether the tyre on the car is flat. Then a quick assessment is made to see whether the spare tyre is in better shape. With COTS software those luxuries may not exist. We might replace quality components with unusable components. So the goal of swapping out a worse component for a better one is not as trivial to achieve with COTS software as with physical systems that use component swapping as their main approach to maintenance.

Besides the problem of knowing *when* to swap out a component for a better one, there is a pot-pourri of other quality problems that cannot be ignored and which in some form affect maintenance:

- *Composition.* The composition problem can best be stated by example. Consider that it is possible to connect two perfectly reliable software components together yet have a less than 100% reliable system. It is also possible to connect two unreliable components and have a 100% reliable system. This is a unique problem to software that does not surface in the mechanical world. What this problem ultimately means is that our compositional reliability theories are not right for software. Therefore, it is not possible to confidently predict a priori how a system will behave from knowledge about how the components behave as stand-alone entities. Can you imagine building a bridge with no knowledge whether it will stand once assembled? Or can you imagine having to take an entire bridge down to replace one girder?
- *The ‘ilities’ Factors.* COTS components can have serious reliability, dependability, security, safety and other ‘ility’ problems (IEEE, 1993, p. 6). For example, we may never know a priori the impact to system-level security from a new component that was swapped in during maintenance. Because components can seriously degrade the ‘ilities’ defined in the system-level specification, methods and tools are needed for rigorously pre-screening components potentially compromising the system-level ‘ilities’.
- *White-box Analysis.* Traditional source-code based (white-box) analyses, such as coverage testing and code metrics, cannot be applied to COTS components by the integrator and user. The publisher can apply these analyses, but that does not mean that they will. Because white-box analyses are vital to attaining higher quality, uncertainty concerning the publisher’s quality control is worrisome.
- *User-profile Testing Assumptions.* Software publishers tend to test their software under assumptions about how their clients will use the product. These assumptions can be seriously flawed and disastrous for ‘outlier’ customers, i.e., those customers that have off-nominal requirements.

- *Common Vulnerabilties.* As companies merge or go out of business, the number of choices in product offerings shrink. This is simply the free-market system weeding out certain products. That translates into all users being vulnerable to the same problems. For example, if a flaw in a Netscape or Microsoft web browser is discovered, then there is a huge set of web users that can now be damaged. This is the reverse of genetic diversity that has allowed stronger species to survive. With COTS, the number of choices is shrinking. For systems used for purposes such as national security, can we tolerate this?
- *Malleability.* Software's malleability is the key justification for building software instead of hardware. As we move toward systems built from software components that are delivered in executable format, and as the percentage of black-box functionality in a system increases, we are shifting toward information systems that are no more malleable than hardware systems.

3. KEY MAINTENANCE CHALLENGES AND THE FUTURE

We have looked at the generic quality problems that are associated with acquired software. It is now time to look more specifically at the key maintenance challenges that face our industry as we try to move toward building information systems from components (that often can be so large as to be considered systems in their own right).

One of the greatest challenges is the problem of what to do if a COTS publisher goes out of business. Most people expect that in such a case the publisher should be willing to release the source code to their licensees. And indeed most publishers are willing to provide this guarantee. But recognize that having unfamiliar code dumped onto a licensee may be an overwhelming experience. After all, the licensee will now have to maintain a source-code base that is foreign to them.

What if a publisher does not go out of business with a product but instead decides to quit releasing new versions? That is, the publisher freezes a product in its current version. If additional product support and upgrades are still needed by licensees, then the licensees will need to either negotiate with the publisher to get the source code or pay what could be substantial customization charges. If neither option works, one additional option is to write wrappers around the component. Wrappers are simply a way of treating symptoms, not root problems. But at least in this case you are assured that the component within the wrapper can remain unchanged,

Both of the aforementioned maintenance challenges are particularly acute for 'long-life' systems. Such systems do not come with plans for decomissioning. In theory, these systems will last forever. We know this will not occur in practice, particularly in an age where one calendar year is joked about as being equivalent to three Internet years. But in our industry, an information system that lasts for 30+ calendar years can be deservedly labelled as having lasted forever.

Another interesting maintenance challenge arises when a COTS publisher provides custom upgrades that are unique for you and that are not in the product line that others are receiving. The problem here is that you no longer have a 100% COTS product. You have a customized COTS product. While this is ideal in the short term, if someday the publisher decides to quit offering customizations, you may find yourself needing the source code because there are no other competing products that provide the custom features your 'one of a kind' COTS software provided. The moral here is to be careful when you are dealing with customized COTS software because you are truly 'painting yourself into a corner'.

One last issue to consider is the maintenance challenge that originates from the percentage of

COTS software in a typical new development. For example, Microsoft's Windows-NT is rumored to be in its entirety around 48 million lines of code. When your application of say 30 thousand lines runs on top of NT, the proportion of functionality that the user is receiving that is attributed to your application may be very small. Microsoft may be providing most of the functionality. If the application fails, however, is it your fault or Microsoft's? Is it the NT portion that failed or the 30 thousand lines you wrote? From your customers' standpoint, they probably do not care. They care that your product failed. Since you are not in the business of maintaining Windows-NT, can you defend yourself? Clearly you cannot anticipate and protect against every defect in 48 million lines of acquired software.

4. DEFENSIVE OPTIONS

So where does all of this leave us? If we do not employ COTS, development costs are unbearable. If we employ COTS, the ability to maintain composite systems is reduced to the level of 'unmaintainable'. Unfortunately, there are not many good options that a system integrator can employ to fight the quality and maintenance problems mentioned above. For example, avoiding COTS altogether and writing your own software functionality is an option but an option that will rarely be viable.

There are several viable options, however, that you should consider in order to protect yourself against acquired software. They include:

1. seeking independent, third-party certification of the COTS product;
2. going to the COTS publisher site and assessing the software directly;
3. performing heavy amounts of system-level testing with the COTS embedded (before actually licensing the COTS software);
4. suing the COTS publisher if their product fails in a manner that causes unreasonable problems for you, the licensee; and
5. building *wrappers* or other middleware around the COTS functionality.

As noted earlier, the main problem with wrappers is that they are simply a way of treating the symptoms of COTS problems and not the problems inside of the components themselves. Further, wrappers can be defective, and every new release of a component can require maintenance of the wrappers.

Note that these options will all incur additional expenses that must be factored into development and maintenance cost estimates, and in fact, depending on the degree to which an option is performed, the costs could be substantial.

5. CONCLUSIONS

Because of the shift toward COTS and component packaging, software maintenance activities are inevitably going to change. I predict that we will spend more time:

1. attempting to increase malleability for inevitable changes in system requirements via component interface languages (e.g., CORBA) and wrapper development;
2. performing component regression testing and performing other activities (such as visiting a vendor's site) in order to pre-qualify acquired components before they are adopted;

3. developing procedures for determining when to accept upgrades;
4. deciding how to maintain source code provided by a software publisher that no longer supports a product; and
5. performing black-box, system-level testing after components are swapped in.

Put simply, we are building unmaintainable systems when we employ acquired software. Whether the cost savings from employing COTS will be so great that these systems can be treated as disposable is unclear. What is clear is that if we are going to move toward a hardware engineering paradigm for maintenance, we must have components that are swapable and that can be trusted to behave as desired by each system into which they are embedded. Where will we get them from?

References

Anderson, S., *et al*. (1997) *Help Wanted: The IT Workforce Gap at the Dawn of a New Century*, Information Technology Association of America (ITAA), Arlington VA, 76 pp.

Baker, S., McWilliams, G. and Kripalani, M. (1997) 'Forget the huddled masses: send nerds', *Business Week*, **3536**(July 11), 110–112, 116.

FAA (1992) 'Software considerations in airborne systems and equipment certification', Document No. RTCA/tDO-178B, RTCA, Inc., Washington DC, 112 pp.

Gilb, T. (1977) 'Spare parts maintenance strategy for programs', *Computer Weekly*, **21**(502), 8–11.

IDC (1998) *Worldwide Software Review and Forecast*, International Data Corporation, Framingham MA, 7 pp.

IEEE (1993) *IEEE Standard for Software Maintenance*, Institute of Electrical and Electronic Engineers, New York NY, 39 pp.

Jones, T. C. (1995) *Patterns of Software Systems: Failure and Success*, International Thomson Computer Press, Boston MA, 292 pp.

Jones, T. C. (1998a) *White Paper: The Impact of Poor Quality and Canceled Projects on the Software Labor Shortage*, Software Productivity Research, Burlington MA, 13 pp.

Jones, T. C. (1998b) *The Euro, The Year 2000, and the Software Personnel Shortage*, Software Productivity Research, Burlington MA, 16 pp.

META (1998a) *Electronic Commerce: A Practical Business Guide*, META Group, Inc., Stamford CT, 96 pp.

META (1998b) *INforum*, META Group, Inc., Stamford CT, 23 pp.

Musa, J. D., Iannino, A. and Okumoto, K. (1987) *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill Book Co., Inc., New York NY, 635 pp.

Perry, W. (1994) *Specifications and Standards—A New Way of Doing Business*, Department of Defense, US Federal Government, Washington DC, 32 pp.

Whiting, R. (1998) 'Chaos study', *Software Magazine*, **18**(12), 20.

Author's biography:

Jeffrey M. Voas is a co-founder and Vice-President of Reliable Software Technologies Corporation, and serves on the Board of Directors for the Center for National Software Studies. He was the General Chair for COMPASS'97 and is the Program Chair of ISSRE'99 and a Program Co-Chair of ICSM 2000. His interests include component software, software certification, information security metrics, software liability, software testing, software dependability metrics and software safety. Jeff has authored more than 85 published papers and co-authored two books. His doctorate is in Computer Science from the College of William and Mary. His email address is: jmvoas@rstcorp.com

